DL 3.66
Marco/9/14
#32

GETTING STARTED

A PC/PILOT program is made up of a series of commands or statements. There are only about twenty different kinds of statements. Each one has a specific purpose. A statement is always written on one line of the screen when writing the program. Each statement starts with a letter called an op code; the op code identifies the particular type of statement. After the op code there is a colon (:), and after the colon there may be some text.

The normal way to write a program is to use a text editor to create a disk file. The disk file consists of the statements that make up the program. Once the disk file is created you can tell PC/PILOT to run the program. We will talk more later about the process of creating the program file on disk.

To get started we will run an already written PC/PILOT program called **SAMPLE1** which is supplied on your PC/PILOT disk. It will allow us to try out one statement at a time without having to write it first into a disk file. To get started type the command

PI SAMPLE1

and push the return key. You can now write your first PC/PILOT statement. Type in the line

T:Hello world!

after you push return you should see

Hello world!

This is an example of a TYPE statement. The T is called the op code. Notice that after the op code you put a colon and after the colon you put the text which was to be displayed by the type

statement. Try putting in another type statement for practice. By experimenting, see what happens if you type the T op code in lower case. Also see what happens if you leave off the colon after the op code.

You should have noticed that it does not matter whether the op code is upper or lower case. You should also have seen that leaving off the colon character after the op code is considered an error by PC/PILOT. In that case PC/PILOT shows you the statement it considers in error, followed by a message that tells what it considers to be wrong. Usually it will be easy for you to tell what is wrong, but if you need more explanation check the ERROR MESSAGES section of the Reference Manual. After any error in a program you can choose to stop the program by pushing ctrl-C (that is holding down the ctrl key while pushing the C key), or you can push any other key to just continue on with the execution of the program.

TRYING OUT GRAPHICS

If you do not have a color adapter on your computer, you should skip this section since you can not do graphics on a monochrome adapter.

If you are not still running the SAMPLE1 program get it started by typing

PI SAMPLE1

Next you can try out the GRAPHICS statement. To do so you must first select a screen mode which allows graphics. Also for convenience you can establish a text viewport at the botton of the screen to keep the things you type isolated from the graphics you draw. To do this type

TS:M4;V0,39,20,24

and push return. The TS op code is used for several purposes; what you did here was to set screen mode 4, and make the last 5 lines of the screen the text viewport. You should have a blank screen now and the cursor should be toward the bottom of the screen.

To draw graphics you tell an invisible turtle which direction to turn and how many steps to take. As the turtle moves it can draw a line of some chosen color. When a program starts, the turtle is in the middle of the screen and is pointed towards the top of the screen. Try the following statement which tells the turtle to go forward 100 steps.

G:F100

You should see a vertical line appear. Now try to tell the turtle to turn to its right and go forward another 100 steps.

G:R90;F100

You should see a horizontal line appear making a right angle with the vertical line. In both the above statements the G is the op code for the GRAPHICS statement. The F100 is a forward command for 100 steps. The R90 says turn right 90 degrees. You can put any number of commands after the colon as long as each is separated from the next one by a semi-colon. Try this statement; it completes the drawing of a box.

G:R90;F100;R90;F100

You can change the color of the lines drawn by the turtle like this:

G:C1

This command sets the line color to color number 1. See the GRAPHICS section of the Reference Manual for information about the various colors you can get. Try drawing a box in the new color:

G:F200;R90;F200;R90;F200;R90;F200

Notice that in the above statement the same sequence is repeated several times. There is a short hand way of representing this on the GRAPHICS statement. Try the following statement, it is equivalent to the last one.

G:*4(F150;R90)

The notation *n(...) lets you repeat any commands in the parentheses n times. This turns out to be a very powerful feature as you will see by trying out the following statements. Two new commands are used below. The E command on the GRAPHICS statement causes the screen to be erased. The L command tells the turtle to turn left. Try each of these statements.

G:E;*3(F350;R120)

C0: amarillo
C1: verde
C2: rojo

```
G:E;*5(F240;R72)

G:E;*8(F200;L45)

G:E;*20(F30;L18)

G:*5(F300;R144)

G:E;C2;*72(F400;R175)
```

There are a few other important facts to know about the GRAPHICS statement. One is illustrated by the following example. Try it to see what happens.

```
G:E;B80;*200(F5)
```

You should see the line run off the right side of the screen and come back on the left side. In general any time you draw off one edge of the screen you come back on the opposite edge. This wrap around nature of the screen makes it act as if it were infinite in all directions.

Actually there are 320 individual dots or pixels, as they are called, across the screen. They are numbered from 0 on the left to 319 on the right. There are 200 pixels from top to bottom; they are numbered from 0 on the top to 199 on the bottom. You can tell the turtle to go to any particular pixel by giving the horizonal or x position and the vertical or y position. Try these statements for an example.

```
G:G20,15;F30

G:G300,100;F25
```

Normally one step for the turtle is one quarter of a pixel. That means that it would have to go at least four steps to make a line two pixels long.You can control the horizontal and vertical

scale factors to set the size of one turtle step. Try the following three statements. They show the effect of the scale factors.

```
G:E;G130,100

G:*3(F40;R120)

G:X4;Y4;*3(F40;R120)
```

Notice that the same triangle is drawn but it is four times as large the second time. The reason is that the X and Y scaling factors are each set to 4. At this setting a turtle step is one pixel on the screen.

There are several uses of the scale factors. One is to compensate for a display screen which has pixels that are not square. For example, if when you draw what should be a square it is a little wider than it is tall, you could adjust for it by setting the X scale to 3 and the Y scale to 4. A second use of the scale factors is to purposely distort a figure. For example, the following statement draws what would normally be a circle, but by setting the scale factors see what happens instead.

```
G:E;X3;Y1;*20(F20;R18)
```

You should see an elipse which is about 3 times as wide as it is high.

At this point you should read the GRAPHICS statement section of the Reference Manual. After having tried these few examples it will make more sense to you than it may have before. You will also notice that there are features described there that were not covered here.

Before you go on to the next section you can clean up the screen by entering

TS:MO

Which takes the screen out of graphics mode and puts it back into text mode 0.

THE USE OF VARIABLES

If you are not already running the SAMPLE1 program then start it by entering

PI SAMPLE1

A variable is a word which names a storage location in computer memory. PC/PILOT has two kinds of variables. The first kind is a numeric variable, which as you might guess, can store a number. The COMPUTE statement can be used to save the result of a computation in a numeric variable. Try for example:

C:X=3+4

To understand what this statement does you might read it as "save the sum of 3 and 4 in the variable X". You can display the value of a variable thus:

T: #X

You should see 7 displayed. Notice that the # itself is not displayed, neither is the X. The # character tells the TYPE statement that a numeric variable follows and that the value of the variable, not the name of the variable, should be displayed. Numeric variables have many uses such as counting the number of right and wrong answers. We have used the variable name X, but you could chose any word that would help you remember what purpose the variable is to serve. Try these three statements which might be found in different parts of a program.

C: WRONG = 0
C: WRONG = WRONG + 1
T:You had #WRONG incorrect answer.

In this case the variable is named WRONG. As a side note, the examples show variable names in

upper case only; in practice you can use upper or lower case interchangeably in variable names. That is, the variable name SCORE refers to the same value as the variable name score.

A second type of variable is the character string, or just string. It can be used to save a list of characters, like a person's name. There are two important facts about a string variable. First, the name of a string variable always has a $ on the end of it. Second, you have to use the DIMENSION statement to set aside memory space for the string before you can use it. Try the following statement. It defines a string variable called NAME$, and sets aside enough memory to store up to 10 characters.

```
D: NAME$(10)
```

The compute statement can be used to set a string variable in much the same manner as a numeric variable. Try these two statements.

```
C: NAME$ = "Mickey Mouse"
T:I like $NAME$.
```

If you did it exactly as shown then you would see a line like

```
I like Mickey mou.
```

Since you set up NAME$ to hold up to 10 characters, only the first 10 characters were saved. Why are there two $ characters, one before and one after the variable? The first $ indicates to the TYPE statement that a sting variable name follows and that the value of the string variable is to be displayed. The second $ is actually part of the variable NAME$.
It would be very helpful at this point to read the sections of the Reference Manual that describe the COMPUTE and DIMENSION statements and the sections on VARIABLES and EXPRESSIONS.

A FIRST PROGRAM

If you are still running SAMPLE1 then stop it by pushing ctrl-C.

To write a program you must create a text file with a name that ends in .PIL ; for this example you should use the name FIRST.PIL . The EZ editor that comes with PC/PILOT makes this process very simple. If however, you already have a text editor you would rather use, go ahead, it does not matter how the program gets into the text file. EZ is described more completely in the Reference Manual. But just to get you started this time, type

```
EZ40 FIRST.PIL
```

You will see a line that says new file, then the screen will go blank. You just start typing the lines you wish. After each line you just push return. If you make an error you can use the backspace or arrow keys to move to the spot in error, then type the correction over the error. When you are done just push the f10 key. Try entering the following simple PILOT program.

```
D:NAME$(10)
T:Hello, what is your name?
A:$NAME$
T:Hi $NAME$, nice to meet you!
```

Once you have typed it in, save it by pushing the f10 key. Next try running the program by typing

```
PI FIRST
```

The first statement sets up a string variable called NAME$ and reserves 10 characters of memory for it. The second statement types out the message Hello etc, which is the first thing

you should see on the screen. The next line contains an **ACCEPT ANSWER** statement signified by the **A** op code. It tells PC/PILOT to allow the student to enter a response. Nothing further happens until the student types something and pushes the return key. Assuming the student types a name, like Mary. The next **TYPE** statement displays a message "Hi Mary, nice to meet you!"

If the program does not run as you expect it to then use the editor to correct the program and try it again. To edit the program with EZ enter

EZ40 FIRST.PIL

exactly the same as you did when creating the program to begin with. You will see your program on the screen. Just move the cursor to the spot you need to fix using the arrow keys. Then type the corrections over the errors.

You should try running the program a few times with different responses. See what happens if you enter a very long name, or no name at all.

Also, if you are using EZ then you should now read the section in the Reference Manual which describes it. Try out each command or function key to make sure you understand what it is used for.

MORE ABOUT THE MATCH STATEMENT

The MATCH statement is one of PILOT's most unique and powerful features. It is used to judge the last student answer against a pattern. The end result of the MATCH statement is simply a YES, the student answer does fit the pattern, or NO, the student answer does not fit the pattern. The YES or NO result can be used to conditionally execute or skip over any statement by appending a Y or N to the op code. As you have already seen, the statement

TY:That is right.

would be executed only if the previous MATCH statement resulted in a YES. Otherwise it would be skipped. Conversely, the statement

TN:That is wrong.

would be executed only if the previous MATCH statement resulted in a NO. Otherwise it would be skipped.

You have considerable latitude in coding the match pattern. First of all, the student answer does not have to match the pattern exactly, it need only contain the pattern somewhere within. That is, a statement **M:AUTO** would give a YES match for a response like **AUTOMOBILE** or **SEMI-AUTOMATIC**. In each case, the pattern AUTO is contained somewhere in the student response.

You do need to consider the distinction between upper and lower case. If you wish to treat answers in either case as equivalent, one way to do so is to use the **PROBLEM** op code to set the **U** option. This forces all student input to upper case as it is stored in the internal answer buffer. Then you would code all match patterns in upper case. This scheme would work equally well with the **L** option and lower case. The S

option can also affect how you code the match patterns since it causes the removal of all spaces from the student answer.

There are several ways you can enhance the match pattern to allow for variations in the student answer. It is very important to do so in order to avoid unnecessary frustration on the part of a student who gives an essentially correct reply which happens to vary slightly from the expected reply. There are two wild card characters you can use, namely the "*" and the "&". The "*" matches any single character in the same position of the answer. For example

M:OL*MPIC would match OLYMPIC or OLIMPIC.

The "&" matches any sequence of zero or more characters. For example
M:RED&BLUE would match RED, WHITE AND BLUE or REDBLUE. One way to think of the "&" is that it means "and". So the statement would mean that the word "RED" and the word "BLUE" must be present in the given order.

Often there are several alternative replies that need to be allowed. The "!" character can be used to separate several possibilites. "!" is often thought of as meaning "or". For example

M:boat!ship

would match either the word "boat" or the word "ship".

The "*", "&" and "!" can be used in any combination within a match statement to give very flexible answer checking. For example:

M:cat&dog!be*r!lion

would give a YES match for "cats and dogs", "grizzly bears", "cold beer" or "lions and

tigers".

Sometimes the match is a little too flexible. Consider the statement M:no which obviously expects a negative answer. If the student were to reply "I don't know" ,it would match since the answer contains "no". In cases where the match needs to be restricted you can use the "%" character. It matches only with a space or the start or the end of the student answer. Some examples will clearify.

M:%no%

would match "no", but not "I don't know" because the first % says the word must not be preceded by a non blank character and the second % says the word must not be followed by a non-blank character. Another example:

M:*ed%

would require a word that ends in "ed" since the * would match any character and the % requires that the ed be followed by a space (or be at the end of the answer.)

The MATCH statement can be told to forgive minor spelling errors by appending the S modifier to the M op code. This will allow a YES match even when the student response differs in some small way from the pattern. This means you can ignore minor mistakes made by the student without deciding ahead of time what mistakes might be made. For example

MS:PILOT

would match with "PILOT", "PILIT", "PILATE", or "PYLOT".

You can use this program to experiment with various MATCH statements. Use it to try the

features of the MATCH.

```
T:Input a reply or ctrl-C to quit.
A:
M: (put whatever you want here)
TY:That matches.
TN:That does not match.
J:@A
```


DISPLAY MODES AND FEATURES

With PC/PILOT you can use the screen in several different **MODES**. The display mode tells how many character rows and columns there are on the screen, what colors appear and whether there may be graphics displayed along with text. In some modes you can have several display pages. A display page is a screen full of information, one of which is currently visible.

The actual colors and display quality you see depend somewhat on the display adapter you have (color or monochrome), and the monitor you have. In decreasing order of display quality, the types of display you can have are: RGB color monitor, composite video monitor and TV set connected via an RF modulator. In general, the things documented here apply only to systems with a color adapter card. If you have a monochrome adapter then you get no color or graphics.

The **TYPE SCREEN** or **TS** statement is used to set the desired screen mode. For example

TS:M4

Sets screen display mode 4.

When a program begins the mode it set to mode 0. The modes are numbered from 0 through 6. Modes 4,5 and 6 allow the use of user-defined characters. In other modes the standard character set is always used. The **GRAPHICS** statement may be used only in modes 4 and 5.

The **NEW CHARACTER** statement can be used to change the apperance of any printable character. All characters are formed on the screen as a block of dots which is 8 high and 8 wide. You define a new character shape by constructing the dot pattern you wish to see with the "." and "/"

characters. The character is displayed with its new pattern any time it is displayed in screen mode 4, 5 or 6. Read the section on NEW CHARACTER in the Reference Manual.

Display modes 0 through 3 provide more than one display page. Normally display page 0 is visible on the screen. To make another display page visible use the TS statement. For example

TS:P2

would make the currently displayed page disappear and make display page 2 appear. Any text displayed would now be placed in page 2. Later if the statement

TS:P0

were executed then page 2 would disappear and page 0 would appear and would look exactly as it had before. One use of display pages would be to set up a frequently used display frame in some specific page by making it the currently visible page then doing TYPE statements to set it up. Once it is set up, change to some other display page. Then, whenever the saved display is wanted it can be instantly flashed onto the screen by changing the page number. This scheme would work well for help menus which may be called for at various points in the program.

The TYPE SCREEN statement also allows you to put text anywhere on the screen you wish to do so. The screen always has 25 lines of text available, the top line is considered line 0, the bottom line is considered line 24. Depending on the display mode there may be either 40 or 80 characters across each line. The leftmost column is considered 0, the rightmost is either 39 or 79. In placing text on the screen you specify a character position as two numbers. The first number is the column number, the second is the

line or row number.

Column and line numbers are used in two ways. First, you can define what is called a **VIEWPORT** on the screen. A viewport is just a rectangular section of the screen in which text can be displayed. When a program starts out, the viewport is considered to be the entire display screen. You can set the viewport by a statement such as

TS:V0,39,15,24

which sets the viewport to be from column 0 on the left, through column 39 on the right and from line 15 on the top through line 24 on the bottom. In a mode with 40 columns this would make the viewport equal to the bottom 10 lines of the screen. When a viewport is set in this way, the cursor is placed at the home, or upper left corner of the viewport. Any text output would display from that point on, and text will be dislayed only within those bounds. If the viewport becomes full it scrolls up one line to accomodate the next line of text. In doing so, all display data outside the text window is left unaltered. You can change the viewport as often as you wish; this facilitates the use of various parts of the screen to display certain data. Each time a new viewport is set up, PC/PILOT remembers what the previous one was. You can get back to the previous viewport by a statement

TS:V;

which sets the viewport and the cursor position in the viewport to whatever it previously was. To see how a viewport works try this program.

```
T:This text is outside the viewport.
T:So it is unaffected by what is
T:Happening in the viewport.
TS:V10,30,11,16
```

```
T:This text is inside the viewport.
T:Type something
T:and I will type
T:it back.
A:
E5:
T:#%B
J:@A
```

You can position the cursor anywhere within the current viewport by a statement like

```
TS:G5,6
```

which tells the cursor to go to column 5 and row 6 of the viewport. Remember, column and row numbering starts at zero. If you specify a row or column too big for the viewport then the cursor is set as close as possible to the specified position while still remaining in the viewport.

The TYPE section in the Reference Manual also describes certain control characters which, if included in a TYPE line can cause the cursor to move.

Normally, text is displayed as if it were single spaced. You can display text double or triple spaced by setting the line spacing option thus:

```
TS:L2
```

which means that from now on one blank line is left between all text lines that are typed (double space).

MODES AND COLORS

The TYPE SCREEN statement can be used to specify the foreground and background text colors. To experiment with this start the SAMPLE1 program by entering

```
PI SAMPLE1
```

Now enter the following lines

```
TS:M1;B2;F1
TX:This is blue on green.
TS:V5,35,10,20;B0;F5
TX:This is in a black viewport.
```

This should show the kinds of things you can do. In general the foreground color is the color of the character itself, and the background color is the color of the square on which the character is displayed. However, there are some differences in the actual effect of the foreground and background colors depending on the display mode you are using. First of all, in the text-only modes (0-3) there are 8 colors to choose from for background or foreground. They are numbered from 0 to 7. On color monitors which implement the brightness signal there can be 8 additional foreground colors (8-15) which are brighter versions of the first 8. For background colors 8-15 you get the same background colors as for 0-7 but any characters written on a background color from 8 through 15 will blink, or flash, on and off. Setting the foreground and background colors in these modes does not affect the characters already displayed on the screen, only the characters subsequently displayed. Since the TX: statement (TYPE with CLEAR modifier on) does the equivalent of writing blanks to the entire viewport, the effect is to clear the viewport to the current background color. If the screen is still as set by the above sequence of statements try the

following three lines to illustrate this point.

```
TS:B4;F0
T:Now using new colors.
TX:Now have cleared the viewport.
```

The screen border, or perimeter of the display screen always has the current background color. Whenever the background color is changed the border color changes immediately.

Modes 4 and 5 are called the **GRAPHICS MODES** because they allow the use of graphics along with text displays. However, the use of color in these modes is different from text-only modes. First of all, there are only four colors available at a time. Three of them (colors 1,2 and 3) are predetermined by the mode. Check the Reference Manual TYPE SCREEN section to see what the predefined colors for modes 4 and 5 are. Color 0 is defined to be the current background color as set by a TYPE SCREEN statement. The background color can be set to one of 8 colors (numbered 0-7). In the graphics modes setting the background color immediately affects the background of all data currently displayed on the screen. This is different from what happens in text-only modes. Also important to note is that in these modes the GRAPHICS statement can be used; and that the line colors (0-3) that can be drawn by the GRAPHICS statement are the same four colors available for text foreground and background.

In all cases you should note that if you choose a background and foreground color that are the same, then any text displayed is invisible. In fact, unless you have a very good color display device then choosing foreground and background colors that are even too similar can have the same result.

Use the SAMPLE1 program to try out various

combinations of modes and colors to see what works well with your own particular system. If you get an unreadable combination you can still enter commands, even though you don't see them. If you get in this state just push return then

```
TS:M0;B0;F7
```

which sets things back to the way they are when the program starts. In fact, whenever you leave a PC/PILOT program you should set the mode back to mode 0 or mode 2 which are the normal 40 and 80 column modes used by DOS.

You may have noticed that the cursor looks different in text and in graphics modes. In the text-only modes the cursor is always displayed and it blinks. In the graphics modes the cursor does not blink and it is only displayed when the program is waiting for the user to type something.

HOW TO CHOSE THE BEST DISPLAY MODE

When designing a lesson it is good to think about the specific type of display that will be used. The results can vary significantly with the type of monitor (RGB, composite video color, TV set with RF modulator or b/w monitor). Modes 0 and 2 give colors similar to mode 1 and 3 on an RGB monitor only. On a composite video or RF modulator these modes are monochrome with good character quality. Modes 1 and 3 give the colors documented in the manual. The color quality depends on the monitor. Modes 4 and 5 give color on all monitors but the character and color quality are excellent only on an RGB monitor. On all others, the colors seem to have fringes of other colors. If you have a color adapter card with a b/w monitor, modes 1 and 3 give very poor character quality. If you have a monochrome adapter card you should use mode 2 only.

ANIMATION

If you are not already running the SAMPLE1 program start it by entering

PI SAMPLE1

PC/PILOT allows for a limited form of animation, or movement of figures on the screen. It can be accomplished by the combination of several commands on the TYPE SCREEN statement. The features, which are described separately in the Reference Manual are as follows. The ANIMATE command allows the display of text by a TYPE SCREEN statement. It allows the text to be displayed across one line or as a block of a few characters on several lines. For example

TS:M0;AHi/Mom

gives a display that looks like this

Hi
Mom

The animate command does not end up moving the cursor from its original position. To move the cursor the WALK command is used. It can be used to move, or "walk" the cursor one space in any direction. For example, try this statement.

TS:A1;WDR;A2;WDR;A3

which should display a "1" then walk down and to the right one space, then display a "2", walk down and to the right, then display a "3".

A third component in making motion is the ability to specify a repetition of several commands. This is done on the TYPE SCREEN much the same way as on the GRAPHICS statement. Try this statement making sure you enter it exactly as shown (no extra spaces).

TS:*30(AX;WR)

You should have gotten a line of thirty X's. Here's why: the *30(...) means to do what is in the the parentheses 30 times. In the parentheses you have an animate command which displays an "X", followed by a walk command which moves the cursor one space to the right. The net effect... thirty X's in a row.

Notice in this next example how inserting a space can cause each previous X to be erased before putting out the next X. First push RETURN then try this statement exactly as shown

TS:*30(A X;WR)

The only difference is the space before the X. But what happens on the screen. Why? The animate command now displays a space, then an X. Then it moves right one from where it started, which puts the cursor under the X. The next time through the loop the space overwrites the previous X (thereby erasing it) and another X is displayed after that. It probably went a bit fast so try it again and watch closely.

If you wish to slow the motion down you can do so by using the DELAY command. For example

TS:*30(A X;WR;D6)

Note the addition of D6 which means delay for a period of six 60ths of a second. By placing the DELAY in the loop, there is a tenth of a second delay between each X.

You may have noticed that the cursor remains visible as the animation procedes. Remember that in screen modes 4 and 5 the cursor is visible only when the user is typing. Try the following two statements which change to screen mode 5

then do the same animation.

```
TS:M5
TS:*30(A X;WR;D6)
```

This looks better since the cursor is not visible during the animation.

It is common to use animation to move characters that have been re-defined by the NEW CHARACTER statement. In fact a figure may be made up of a block of several re-defined characters to make a shape bigger than one character cell. This example shows a block that looks like

```
WX
YZ
```

being moved, but the characters used could be any four re-defined characters you wish. Also the block could be any number of characters.

```
TS:*20(A WX/ YZ;WR;D5)
```

Or to move it right to left:

```
TS:*20(WL;AWX /YZ ;D5)
```

In these as in other animation sequences the placement of spaces in the animate command is important to avoid leaving a trail behind on the screen.

USING COMPUTE AND EXPRESSIONS

PC/PILOT does computations in a manner similar to other programming languages such as BASIC. For example, to do a computation and assign the result to a variable the statement would look like this.

```
C: X = ( Y + 10 ) / 3
```

C: is the op-code for the COMPUTE statement. Everything to the right of the "=" is called an expression. In this case, 10 is added to the value stored in variable Y, the result is divided by 3 and that result is stored in variable X.

A second context in which an expression is useful is as a conditional, such as

```
J(X < 10):LOOPA
```

In this case the expression X < 10 is evaluated as either -1 if true or 0 if false. If the expression is true then a JUMP is taken to the label LOOPA. If the expression is false the JUMP statement is skipped and the control passes to the statement after the JUMP.

Use EZ40 or another editor to enter the following program as file LOOP.PIL.

```
R:Sample loop program
C: I = 0
*BACK T: #I
C: I = I + 1
J(I <= 10):BACK
T:Done
```

Now run the program by entering

```
PI LOOP
```

You should see the numbers from zero to ten displayed. This is an example of what is usually called a loop. It consists of the following general steps.

(1) Set a counter variable to an initial value
(C: I = 0)
(2) Define a label at the start of the loop
(*BACK)
(3) Do the statements that are to be in the loop
(T: #I)
(4) Set the counter variable to its next value
(C: I = I + 1)
(5) Test the counter against a limit and jump back to the loop start label if not yet at the limit
(J(I <= 10):BACK)

In computing expressions there are a number of operators which can be used. They include the expected things like add(+), subtract(-), multiply(*) and divide(/). These operators do the computation to an accuracy of about 6 places. In computer jargon the computations are done with single precision floating point math. There are also all the various comparison and logical operations such as less than(<), greater than(>), equal to(=), and(&), or(!) etc. These operations always return a 1 for true or a 0 for false. You can refer the the Reference Manual section on OPERATORS for more information.

Also provided are some FUNCTIONS. An example of a function is

C: A = INT(X/2)

In this case INT is a function. It gives the value of whatever is in parentheses truncated to an integer. For example, if X has a value of 7, then X/2 gives 3.5, and INT(3.5) is 3. So the variable A is set to 3. There are many other

functions described in the section on FUNCTIONS that are not discussed here.

However, there is one more function worth mentioning here. It is the KEY function. The KEY function tells whether or not a key has been pressed on the keyboard. To understand its utility you have to consider how the ACCEPT statement works. Normally, whenever keyboard input is expected from the student, an ACCEPT statement is executed. The ACCEPT statement waits for the student to enter a reply and push return. Even the ACCEPT SINGLE statement waits until at least one key is pushed. Now, consider a program in which the action should not stop to wait for an input. Perhaps the program is a simulation game in which the student must respond in response to something changing on the display. In this case if the program executes and ACCEPT statement then everything waits until the student replies. The solution is to use the KEY funtion to periodically test for a depressed keyboard key. If there is none depressed then the program can go on with whatever else it is doing; if a key is depressed then the KEY function tells what the key is and the program can take the appropriate action for that key. The following program shows the essential elements of this sort of logic. Use an editor to put it on disk as file KEYS.PIL.

```
R:sample use of KEY function
C:I=500
*XYZ
C: K = KEY(0)
TH(K):#K
TH:.
C:I=I-1
J(I):XYZ
```

When you run this program the screen will begin to fill up with dots. While it is printing the dots just push a key here and there on the

keyboard. Whenever you do you see a number printed. Try at a couple of times then read the notes below. (To run it enter PI KEYS)

Notes on the above program

There are several interesting points illustrated in this program. First you should recognize the loop construct with the variable I used as a counter. In this case I counts down from 500 to 0. The last two statements of the program cause I to be decremented by 1, then the JUMP is taken based on the expression "I". Since an expression is considered false if it is zero, or true if it not zero, the JUMP is taken each time through the loop until I becomes zero. When I becomes zero the JUMP is not taken and the program ends.

The statement TH:. displays one period each time it is executed. The H modifier on the TYPE statement says not to put the cursor on the next line after the TYPE. That's why the dots fill across each line.

The statement C: K = KEY(0) sets the variable K. It is set to zero if no key has been depressed. If a key has been depressed then K is set to a number which represents the key value. For printable characters the number is the position in the ASCII table of the character. For example, a space is 32, a 1 is 48, and A is 65. You can refer to an ASCII table, such as the one in your BASIC manual for other characters.

The statement TH(K):#K is another TYPE statement with the HANG modifier. It is conditioned by the expression (K) which means if K is not zero then execute this statement. Since K is zero if no key is depressed then this statement is skipped unless a key is depressed. If a key is depressed then the numeric value of the key is displayed.

This program is also useful to see what values

you get for the various control and function keys on the keyboard. Try the program again; this time push some of the control and function keys.

ARRAYS

PC/PILOT allows for arrays or lists of numeric values. Before a variable can be used as an array, space must be set aside for it by the DIMENSION statement. For example

D: ABC(20)

establishes an array named ABC with 21 entries. The first entry is refered to as ABC(0) and the last one is refered to as ABC(20). To refer to any particular element of an array, you enclose a subscript in parentheses after the array name. The subscript can be any expression which evaluates to a number from 0 to the defined upper bound of the array. Key in the following program and try it to see an example of array use. Call the program ARRAY.PIL .

```
R:sample array program
D:A(10)
R:loop to fill array with squares
C:I=0
*LOOP1 C: A(I) = I * I
C:I=I+1
J(I<=10):LOOP1
R:loop to display the array
C:I=0
*LOOP2 T:   #I   #A(I)
C:I=I+1
J(I<=10):LOOP2
T:bye
```

What would happen if the the DIMENSION statement were changed to D:A(9) ? Make this change to the program an run it again. The program should stop two times with error messages. You can look up the error messages in the Reference Manual to see what they mean.

STRINGS

Besides doing computations with numbers you can manipulate strings of text. To do so you can use string variables. A string variable is distinguished from a numeric variable in that a string variable always has a $ appended to the end of the variable name. For example **NAME$** . Before a string variable can be used to store a string value space must be set aside for it by the DIMENSION statement. The statement

D: Z$(100)

establishes a string variable called Z$ (pronounced "zee dollar") and sets it maximum length to 100 characters. At any given time Z$ can store from zero to one hundred characters. The number of characters stored in Z$ is called its length . After the DIMENSION statement Z$ has no characters stored in it so its length is zero. A zero length string is sometimes refered to as a null string. There are several ways to get a value stored in a string variable. One way is via a COMPUTE statement as

C: Z$ = "This is a string value."

A second way is as an argument on an ACCEPT statement. This puts the student reply in the string.

A: Z

Whenever a new value is stored in a string variable then string variable's length is set equal to the number of characters assigned to the string variable. The LEN function returns the current length of a string. For example

C: L = LEN(Z$)

sets variable L to the number of characters in

variable Z$.

If a string which is longer than the dimensioned length of a string variable is assigned to the string variable then the string value is truncated (chopped off) on the right end to match the maximum length of the string variable.

You have already used the string variable to save the student name and include it in feedback messages to the student. There are many other uses of strings. Suppose you wish to remove all commas and periods from a student answer. The following sequence would accomplish this.

```
D:XY$(80)
A:
C: %B = SWP(", ")
C: XY$ = SWP(". ")
C: XY$ = RSP(XY$)
```

The first line sets up string XY$. The second line accepts a reply and stores it in the answer buffer (%B). The third statement changes all commas in the answer buffer to spaces and stores the result back in the answer buffer. The fourth statement changes all periods in the answer buffer to spaces and stores the result in string XY$. The last line removes spaces from the string XY$ and places the result in XY$. The entire effect is to set XY$ to the student answer with all commas and spaces removed. If the result is then to be judged by MATCH statements then it could be put back in the answer buffer by a statement

```
C: %B = XY$
```

The system variable %B is called the answer buffer. It is where the ACCEPT statement saves the student answer. It can be used in an expression as if it were a string variable with

maximum length 80 characters.

There are several string functions and one string operator. The string operator is concatenation. Its purpose is to join two strings together into one string. For example

```
C: A$ = B$ !! C$
```

sets string A$ to a value equal to what is stored in B$ followed by what is stored in C$.

A substring or piece of a string can be used by supplying subscripts after the string name. There are two allowable forms of string subscripts. The first is demonstrated here.

```
C: A$ = B$(4)
```

This statement sets string variable A$ to the one character at position 4 of string B$. It does not change B$. When computing character positions, the first character in a string is considered to be position 1. The second form is

```
C: A$ = B$(5,3)
```

Which sets A$ to the 3 characters found starting at position 5 of B$.

When subscripting strings, the subscripts can not specify a substring that is beyond the current length of the string variable.

It is also possible to assign to a substring as:

```
C: A$(4) = "X"
C: B$(5,3) = "ABC"
```

In these cases the assigned value is truncated or padded with spaces on the right if it is not the correct length for the substring.

STUDENT RECORD KEEPING

PC/PILOT can keep a file of student records called K.REC. You have complete control over what data is kept in the file. One simple example is shown here. The example shows how to save the student's name and a score when the student completes the lesson. Let's assume that the student's name is placed in the string variable NAME$ at the start of the program by statements that look like this.

```
D:NAME$(20)
T:What is your name please?
A:$NAME$
```

Also assume that as the program runs, two numeric variables are set; the first, called TRY, is set to zero at the program start and incremented by one each time an answer is accepted. The second, called RIGHT is also set to zero at the program start, but is incremented by one each time the student inputs a correct answer. Thus at the program start would be

```
C: TRY = 0
C: RIGHT = 0
```

and after each ACCEPT statement which accepts an answer attempt the line.

```
C: TRY = TRY + 1
```

and after each MATCH for a correct answer

```
CY: RIGHT = RIGHT + 1
```

Then, when the student reaches the end of the lesson, the variable TRY has the number of attempts made and the variable RIGHT has the number of correct answers given.

To get a record placed on the file K.REC the

KEEP statement is used. In this case

```
K: NAME$ !! "-" !! TRY !! "-" !! RIGHT
```

causes a line with the students name and the two numbers to be appended to the end of the student records file. The record might look like this

```
Mary White-25-15
```

The sample KEEP statement shown creates the line by using the concatenation operator (!!) to combine the name, a dash, the number of tries, a dash and the number correct. This expression takes advantage of the fact that PC/PILOT automatically converts the numbers TRY and RIGHT to strings to suit the context of the concatenation operator which expects string arguments.

Once out of the PC/PILOT program the file K.REC can be examined by entering the DOS command

```
TYPE K.REC
```

or by editing K.REC with a text editor. If you wish to compute statistics from the data in the file it could be done with a BASIC program, or another PC/PILOT program which reads the file.

The K.REC file continues to accumulate records until it is explicitly erased by the DOS command

```
ERASE K.REC
```

There are many other possible ways to use KEEP. For example, when you are first testing a lesson on students you may wish to save every response entered by a student so you can later make improvements based on the experience. A simple way to do this is to place a KEEP statement after every ACCEPT statement. The keep statement

might look like this.

K: "1-" !! %B

This line puts a "1-" followed by whatever is in the answer buffer (%B) to the K.REC file. The "1-" is simply to identify which ACCEPT statement in the lesson is being answered. The next ACCEPT would be followed by

K: "2-" !! %B

an so on. Later when you TYPE K.REC you see what ACCEPT statements were replied to and what the replies were.

USING SUBROUTINES

Sometimes a section of a program performs some logical function that is useful in more than on context. In this case it is helpful to make tha program section what is called a subroutine. subroutine always begins with a label and end with an END statement. Usually a subroutine i placed within a program in a place where th normal flow of control will not just "fall int the subroutine, so the only way to get to it i by a reference to its label. The USE statemen works in a manner similar to the BASIC langua GOSUB statement.

Once a subroutine is included in a program th it can be used by other parts of the progra whenever its specific action is desired. The U statement calls a subroutine as follows:

U:label

USE is much like JUMP in that control passes the statement after the designated label. T difference is that with USE, PC/PILOT remembe where the USE statement is by saving i location on something called the USE STACK. T when an END statement is executed by t subroutine, PC/PILOT can pick up the ret location from the USE STACK and come back to statement after the USE.

Subroutines can be nested; that is, subroutine can USE another subroutine. E subroutine returns to its caller by execut and END statement. Key in this program USEIT.PIL and run it.

```
T:first
U:sub1
T:second
U:sub2
T:third
```

(continued on next p

```
PR: E
J: overit
*SYSX C: $B(1) = " "
X: $B
E: @A
*overit
```

This sequence takes advantage of the ESCAPE option which works like this. The PR: E enables the ESCAPE command to be used. Later in the program, any time an ACCEPT statement is executed PC/PILOT checks the first character of the answer for a "@" character. If the "@" character is found then PC/PILOT calls the subroutine SYSX as if a statement U:SYSX were executed.

The purpose of this sequence is to supply a SYSX subroutine which contains 3 statements. (The J: overit statement insures that the SYSX subroutine is not executed until it is called for.) The first statement C: $B(1) = " " changes the "@" to a space. The second statement executes the answer buffer as a statement. The third statement ends the SYSX subroutine and goes back to re-execute the last ACCEPT statement (the one from which we came).

Here's how you use it when you are testing your lesson program. When the program starts the SYSX subroutine is skipped over because of the JUMP. Each time your lesson does an ACCEPT statement you can do one of two things. You can just answer the lesson in the normal way in which case nothing special happens. Or, you can put in a PC/PILOT statement, preceded by the "@" character. For example, if you entered

```
@T:#X #Y #Z
```

you would see the values of variables X,Y and Z at that point in the program. Then you could answer the question in the normal way since you

would be right back at the same ACCEPT statement after the variables are displayed. If you wished to change the value of variable X in the program you could enter

```
@C: X = 5
```

This allows you to set up to test certain situations in the program. If you wish to jump to another location in the program you could do so by entering

```
@E:label
```

The reason for E: rather than J: is that since the statement you put in is executed in a subroutine, you would want to end the subroutine before jumping off to some other point. The END statetment with a label does just that. It should be pointed out that the GOTO command could be used for the equivalent action. To use it the first statement in the sequence shown should be changed to

```
PR: EG
```

This enables both the ESCAPE command and the GOTO command. Then, to jump off to another place in the program you would just enter

```
GOTO label
```

A second handy use of EXECUTE INDIRECT is to save a GRAPHICS or SOUND statement which is to be used repeatedly in a string variable. Then to repeat the GRAPHIC or SOUND statement only the string variable needs to be used. Enter and run the following sample program as file EXIN.PIL to try out this use.

```
ts: m5
d: cir$(50)
c: cir$ = "g:*18(f30;r20)"
c: i = 40
*many xi:cir$
g:f35;r9
c: i = i-1
j(i): many
w:150
ts:m0
e:
```

The variable cir$ is used to contain a GRAPHICS statement that draws a circle. Try changing the third line of the program to

```
c: cir$ = "g:*8(f60;r45)"
```

then run the program again.

The next section shows another good use of EXECUTE INDIRECT. As is typical of this statement, a whole program scheme is built around the capability even though there is only one EXECUTE INDIRECT statement in the program.

RANDOM QUESTION SELECTION

One very typical technique in a CAI program is to give the student a random set of problems chosen from a large set of possibilities. This makes the lesson slightly different each time it is run. The following program outlines one method for doing this. The actual question frames are not here, you could fill in with whatever questions you wish. What is shown here is how to randomly select a group of the questions.

First, a few assumptions: suppose that the student is to be given 10 questions out of a possible set of 30 to chose from. Each question begins with a label of the form

QUESnn

where "nn" is a number from 1 to 30. Suppose also that the student must not be given the same question twice in one sitting. And when the student has completed the 10 questions his name and the number of correct answers on the first try are to be stored in the student records file. The program would start off something like this.

```
R: Q$ is used to keep track of
R: which questions given so far.
D: Q$(30)
R: next two lines set Q$ to 30 0's
C: Q$ = "0000000000"
C: Q$ = Q$ !! Q$ !! Q$
R: N is used to count questions done
R: C counts number right on first try
R: NAME$ is student's name
D: NAME$(20)
C: N = 0
C: C = 0
```

(continued next page)

```
T:For the record,
T:...what is your name?
A: $NAMES
T: You will be doing 10 problems.
T: Ready?
A:
R: here to select another problem
*NEXT
R: P is problem number selected
R: next line gets a number from 1-30
*AGAIN
C: P = RND(30) + 1
R: next line jumps back to try again
R: if question previously used.
J(QS$(P)="1"): AGAIN
R: next line shows this one used up
C: QS(P) = "1"
R: next line calls the question frame
XI: "U:QUES" !! P
R: count one more done
C: N = N + 1
R: jump back to do more if not done
J(N<10): NEXT
R: 10 questions done, save results
K: NAMES !! C
R: next line returns to DOS
E:
```

As you can see, the program is heavily laced with REMARK statements. Remarks like these don't affect how the program runs but they will make it much easier for you to remember later how your program was intended to work. As an absolute minimum you should always include remarks that state what all the variables are used for and should prefix each logically distict section of a program with a few general remarks about what the section does.

The above program section would be followed by thirty question frames. For things to work properly each question frame would have to

follow a few rules. First, the question frame must begin with a label like QUES1, QUES2, QUES3 and so on up to QUES30. If you wish to use numbers other then 10 questions out of 30 possible it should be very easy for you to make the appropriate changes to the above program. Notice that the question frames are invoked by a USE SUBROUTINE statement. This implies that each question must end with and END statement. The END statement returns control to the statement immediately following the EXECUTE INDIRECT. Also, for the variable C to be set properly, each question would have to add one to C if, and only if, the student replies correctly on the first answer attempt. The student's name is available in the variable NAMES$ if a question frame needs to include it in a message. The variable N should not be changed within a question frame since it is used to count how many questions have been completed.

An appropriatly coded question frame might look like this

```
*QUES3
TX: In four-four time, how many beats
: does a HALF NOTE get?
A:
M: 2!two
TY: Very good.
CY1: C = C + 1
EY:
M:1!one
TY:No, that would be a quarter note.
M:3!4!thre!four
TY:No, not that many.
T1:Try again.
T2:Hint: it's twice what a
:quarter note gets.
T3:The answer is 2 beats.
E3:
J:@A
```

This question frame presents the question, accepts an answer and checks to see if it is correct. If it is correct the message "Very good." is displayed. In addition, if it is correct and it is the first attempt then 1 is added to variable C (CY1: C = C + 1). Then, a return is made to the main loop after a correct answer via the EY: statement which says END if the MATCH was YES. After that there are some feedback messages for anticpated incorrect answers and some succesive hints for successive incorrect replies. On the third incorrect reply the student is given the answer and the question is ended by the E3: statement.

Create a program file named QUEST.PIL and enter in both the program start section and the sample question given. Rather than make up 30 questions just to try out the program you can try it out with the one question by changing the EXECUTE INDIRECT statement to be the following instead.

```
U:QUES3
```

This would have the effect of choosing this one sample question every time. Make this change to the program file and run it.

This program can serve as the outline for other programs you might create in the future.

MULTIPLE PROGRAM MODULES

CAI lessons often tend to be very long files. There are several good reasons to construct a large lesson as a set of smaller modules. One reason is that very long files are not convenient to handle with most text editors. Another is that constructing a large program out of smaller logical subdivisions results in a more structured program. This facilitates later additions and changes to the program. With PC/PILOT each module of a program is created as a text file with a suffix of .PIL in exactly the same way as an individual program would be.

One PILOT program can pass control to another by the LINK statement. It has two possible forms as shown here.

```
L: PARTS

L: MAIN,NEXT
```

The first example above links to a file PARTS.PIL and begins execution at the first line of that file. The second example links to file MAIN.PIL and starts execution at the label NEXT within that program.

LINK works in a manner similar to CHAIN in BASIC.

All variables, arrays and strings are remembered across a LINK and can be continued to be used by the linked module.

MODIFIERS

A modifier is a letter appended to an op-code. It causes some change in the way the statement is executed. The valid modifiers are shown below. Each is described with the corresponding op-code.

TS:	type screen
AS:	accept single
MS:	match spelling
AJ:	allow type ahead
MJ:	auto jump to next match if no
PJ:	machine level OUT instruction
AX:	accept with default keyboard values
TX:	type clear
GX:	graphics image
FX:	file open/close
DX:	allocate string on 16-byte boundary
NX:	redefine input key value
PX:	absolute memory POKE

If a modifier (J, S or X) is coded on any other op-code it is ignored.

CONDITIONAL

A conditional is a letter, digit or expression appended to any op-code. The conditional causes the statement to be executed only if the specified condition is true. If it is false then the statement is skipped. Each conditional is shown below as coded on a TYPE statement, however, they work the same on any statement.

TY:	type if last match was YES
TN:	type if last match was NO
TE:	type if error flag is true
T(exp):	the expression is evaluated, if it is true (non zero) then the statement is executed, otherwise it is skipped.
TC:	type if previous expression was true
Tn:	n may be a number from 1 to 9. Type if n is equal to the accept counter. The accept counter is a value which tells how many times the last accept has been executed in a row.

Any number of conditionals may be appended to the op-code. All must be true for the statement to be executed.

EXAMPLES:

```
TY: Correct

T(count+2 < LIMIT): No, look at this.
JC: REVIEW

TN3: That's your third wrong try.
```

See also: MATCH, ACCEPT


```
A:
A: $variable #variable
AJ:
AS:
AX:
```

ACCEPT is used to input a student response. The response may be as long as 80 characters. The actual maximum response may be set by the A option of the PROBLEM statement. The student types her response, using backspace as necessary, followed by a return key. The response is automatically editted per the options set on the last PROBLEM statement. The editted response is then placed in the system variable %B, also called the answer buffer. If the X modifier is not included then the values input for each key on the keyboard may be affected by previous NX: statements. If the X modifier is included on the ACCEPT then the input values of all keys are the default standard regardless of any previous NX: statements.

Input characters in the range of ascii 32 to 255 are accepted, see appendix A for a listing of the standard character set.

Normally, the student can not use type-ahead. That is the answer can be keyed only when the ACCEPT statement is actually reached. However, the J modifier (AJ:) allows type-ahead to be used for this ACCEPT.

The S (single) modifier accepts one keystroke only. No editting of the value is done; it is placed in the answer buffer as a one character string. AS: can be used to detect function or cursor keys.

One or more string or numeric variables may be

included on an ... variable, the value of the answer buffer is assigned to the string. For each numeric variable, the first number found in the answer buffer is assigned to the variable. If no number is found, the E conditioner is set to true.

If enabled by the last PR: statement then the student can cause a program jump by entering "GOTO destination".

If enabled by the last PR: statement then the student can cause the equivalent effect of "U:SYSX" by beginning the response with the "@" character.

EXAMPLES:

A:

A: #X
TE: I need a number. Try again.
JE: @A

AS:
T: The key value is #(ASC(%B))

See also: PROBLEM, NX:, GOTO AND ESCAPE

6

COMPUTE - assignment statement

C: target = expression

The expression is evaluated and assigned to the target. The expression may yield a number or a string. The target may be a simple variable, a subscripted array variable, a string variable, a subscripted string variable, or the system variables %B or %A. Since %B is considered to be a string variable it can be subscripted. If the type of the expression does not match the type of the target it is automatically converted from string to numeric or numeric to string to match the target type.

When assigning to an unsubscripted string variable the value is truncated to the dimensioned string length if it is too long. Then the string variable is assigned the value and it takes on the length of the value. When assigning to a subscripted string the value is truncated or blank-padded on the right to fit the substring. The length of the string variable does not change.

EXAMPLES:

C: X = A + B

C: NAME$ = "LARRY"

C: T$(50,13) = "UNITED STATES"

C: TABLE (I) = RND(100)

c

DIMENSION - allocate arrays and strings

D: variable (size),....
DX: variable (size)

DIMENSION creates an array of numbers or a character string. If the variable ends with the "$" character it is a string, otherwise it is an array. The amount of array and string space available is dependent on the amount of memory installed up to a usable limit of 128K. A character string is allocated space to hold a string up to the specified size. It is initialized to length zero. A variable cannot be used as a string until it has been dimensioned. A numeric array is allocated space for size+1 real numbers. The first position in a numeric array is subscript zero. Array elements are not initialized to any particular value.

If a string variable is to be used to contain a machine language subroutine (see V: statement) then use the DX: op code to insure that the string is allocated on a 16-byte boundary for compatibility with segment register limitations.

EXAMPLES:

```
D: X(20),Y(30)

D: NAME$(50)
```

See also: SUBSCRIPTING

d

END - terminate subroutine or program

E:
E: destination

If there is no unended USE in effect the EN causes the termination of the program and return to DOS. If there is an unended USE i effect then END pops up the USE stack by on entry. Then if there is no destination on th END a return is made to the statement after tł corresponding USE. If there is a destination c the END then the return is made to tha destination rather than to the statement aft the USE.

See also: USE

EXAMPLES

```
E:

E: NEXT

*QUITE:QUIT  (always returns to DOS)
```

FILE - access external files

F: filename
FX:
FI: position-expression,variable$
FO: position-expression,value-expression

The FILE statement allows access to disk files. It can also be used to access the printer or auxiliary port. Only one external file may be open at a time. The first form shown above closes any previously opened file then opens the named file for access. If the filename does not exist it is created automatically. The special file names PRN: and AUX: are used to open the printer or RS232 port for output. The FX: without a file name just closes any open file.

FI: positions the file to the relative byte specified by position-expression then reads data into the specified string variable. The number of bytes read is the dimensioned length of the string variable. If an attempt is made to read beyond the end of file the string variable is padded with CHR(255) characters.

FO: positions the file to the relative byte specified by position-expression then writes the string value of the value-expression. The number of bytes written is equal to the length of value-expression.

The position-expression is ignored for files PRN: and AUX:. No automatic appending of control characters is done. If you wish to write returns, line feeds, etc., put them in the value-expression. See also AUX function.

EXAMPLES:

```
FX: DATAFILE.XYZ
FI: 100, D$
FO: ILOC, NAMES$ !! SCORE
```


R ✓

GRAPHICS - display graphic images

G: command-list
GX: filename
GSX: filename

The GRAPHICS statement works only in screen modes 4 and 5. The first form allows the execution of a list of turtle graphics commands to draw lines and shapes. The second form reads a previously saved graphics image from a disk file and displays it on the screen. The third form saves the current screen display in the specified disk file. The disk file is 16K bytes and corresponds to an exact bit map of the color adapter display memory. (see GIE manual)

A list of turtle graphics commands consists of one or more of the following commands, separated by ";" characters. Turtle graphics consist of telling the (invisible) turtle which direction to head and how far to walk. As the turtle walks it can leave a colored line on the screen. The turtle's heading ranges from 0 degrees, or straight up, to 359 degrees with the heading angle increasing as it rotates to its right. If the turtle walks off the screen it comes back on the opposite edge. In the command description below n represents a numeric variable or an integer constant.

Hn	set turtle heading to n degrees
Rn	rotate turtle heading right n degrees
Ln	rotate turtle heading left n degrees
Fn	walk forward n steps
Gx,y	set turtle location to pixel x,y
Dx,y	draw a line to pixel location x,y
Cn	set line color for F or D to n (0-3
E	erase screen to background color
*n(...)	repeat the enclosed commands n time
Xn	set horizonal scale factor
Yn	set vertical scale factor

In modes 4 and 5 the screen is 320 pixels across by 200 down with valid line colors 0-3. Color 0 selects the current background color as set by TYPE SCREEN. Colors 1, 2 and 3 are determined by the current mode as selected by the TYPE SCREEN statement.

	Color 1	Color 2	Color 3
MODE 4	GREEN	RED	YELLOW
MODE 5	CYAN	MAGENTA	WHITE

Adding 8 to the color number causes the color to be exclusive ored with the existing color. Possible background colors (COLOR 0) are documented under TYPE SCREEN.

The X and Y scale factors can be adjusted for a particular display to achieve uniformity in horizonal and vertical step sizes. The default value is X1;Y1 , which makes a turtle step size equal to one quarter of a pixel. To make a step equal to a pixel set the values to X4;Y4.

EXAMPLES:

```
G: *18(F60;R20)          draws a circle

G: *4(F80;L90)           draws a box

G: C2;*5(F160;R144)      draws a red star

G: X6;Y3;*36(F10;R10)    draws an elipse

G:G100,200;D150,85       draws a line from
                           (100,200) to (150,85)
```

See also: TYPE SCREEN

JUMP - goto a destination

```
J: label
J: @A
J: @P
J: @M
```

JUMP transfers control to the specified destination. The destination may be a statement label. In this case do not put the "*" before the label on the J: statement. The @A destination causes a jump back to the last ACCEPT executed. The @P destination causes a jump to the next PROBLEM. The @M destination causes a jump to the next MATCH.

EXAMPLES:

```
J: @A
J: NEXT
```

See also: STATEMENT LABELS

KEEP RECORDS - save data on disk

K: expression

The KEEP statement writes the expression value to the end of the file K.REC. If K.REC does not exist it is created automatically. A new-line sequence is appended to the string expression as it is written to the file so the resultant file is a standard ascii text file which can be listed or processed by an editor. The K.REC file continues to grow until it is explicitly erased by a DOS command.

EXAMPLES:

```
K: "PART3"!!NAME$!!RIGHT
```

```
*Q3 A:
K: "Q3"!!%B       saves student response
```


chain to another program file

LINK - chain to another program file

L: filename
L: filename, destination

LINK closes the current PC/PILOT program file and transfers control to filename.PIL. If the optional destination is given then a JUMP is automatically made to that destination in the new program.

EXAMPLES:

```
L:CHAPTER2

L:CHEMX,Q3
```

See also: JUMP

√MATCH - compare student response

M:pattern
MS:pattern
MJ:pattern

MATCH compares the student answer buffer (%B) with the pattern. The result is a YES or NO match which can be tested by the Y or N conditionals. If the S (spelling) modifier is given then the match is YES even if the student made minor spelling errors. If the J (jump) modifier is given then upon a NO match an automatic JUMP is made to the next MATCH statement.

If the pattern is found anywhere within the answer buffer then a YES match results. The following special characters may be embedded in the pattern.

*	matches any one character
&	matches any string of zero or more characters
!	separate alternative patterns
%	matches only a space or the start or end of the answer buffer

The result of the match can be tested by the Y or N conditionals.

(continued next page)

EXAMPLES:

statement	response	result
M:HAT	HAT	YES
	CHATTER	YES
M:%HAT%	HAT	YES
	CHATTER	NO
M:CAT&DOG	CATDOG	YES
	CATS AND DOGS	YES
	DOGS AND CATS	NO
MS:GREEN	GREAN	YES
	GRENE	NO
MS:CAT!DOG	CAT	YES
	DOG	YES
M:H*T	HOT	YES
	HAT	YES
	HT	NO

See also: ACCEPT, JUMP, CONDITIONALS

K

NEW CHARACTER - redefine character pattern or input key value

N:c

```
...//...
..////..
.//////.
...//...
...//...
...//...
...//...
```

NX: x,y ; x,y ...

The **N:** statement changes the 8 by 8 dot pattern for any of the 128 changeable characters. The changeable characters are the characters from 128 to 255 in screen modes 4,5 and 6. Prior to execution of any N: statement the characters from 128 to 255 have unpredictable display patterns in modes 4,5 and 6. Once at least one N: statement has been executed, the characters from 128 to 255 that have not been explicitly changed will have the same display pattern as the corresponding character in the range 0 to 127. The N: statement has no effect on the characters displayed in screen modes 0-3. In those modes, the characters displayed are always as shown in appendix A. The letter c shown in the format of N: above stands for any character from 128 to 255. The EZ editor provides a way to enter characters in that range into a program. It is followed by 64 periods and slash characters. They depict the pattern of zero (.) and one (/) bits that will be used to display the character. The 64 bits may be given on 8 lines of 8 bits each to give a visual picture of the character pattern. Or, the 64 bits may be given all on one line, or any desired format that adds up to 64. Intervening spaces are ignored.

The redefined characters are used only in screen

modes 4, 5 and 6. In other modes the standard character set is always used.

The **NX:** statement may be used to change the code generated by any key on the keyboard. The normal codes generated are shown in appendix A. The number x shown above stands for the normal default value of a key, the letter y stands fo the new value to be generated by the key. I must be in the range 0 to 255. Any number o keys may be redefined in one NX: statement. Onc a key has been redefined it affects all ACCEP statements except those with the X modifier. Th X modifier on an ACCEPT statement causes any ke re-definitions to be ignored for that ACCEPT.

EXAMPLES:

```
R: define # to look like a box
N:# //////// /....../ /....../ /....../
   /....../ /....../ /....../ ////////


R: cause key f1 to generate code 225 (Beta)
NX: 187,225

R: cause upper case A to give a-accent
NX: 65,160
R: restore upper case A to normal
NX: 65,65
```

See also: TYPE SCREEN, ACCEPT

PROBLEM - set options

PR: list

PROBLEM has two uses. The first is to provide a destination point for a J:@P statement. In this case PROBLEM can be put at the beginning of each frame or question. The second use is to specify one or more lesson options. The list consists of zero or more of the option letters below. If no letters are given then all options remain as previously set. Otherwise each option is set if the corresponding letter is present or reset if the letter is omitted.

U	if set, all student responses are translated to upper case
L	if set, all student responses are translated to lower case
S	if set, all spaces are removed from student responses
G	if set, the GOTO command is enabled
E	if set, the ESCAPE command is enabled
W	clears the label table and forgets about all labels passed so far. Thus it is impossible to jump backward. This allows the same labels to be used again subsequently.
An	sets maximum ACCEPT response length to n, n may be a number from 1 to 80. If n<1 or n>80 then it is set to 80
Z	if coded alone, causes U, L, S, G and E options to be reset

Note: The "R" of "PR:" may be omitted. It is allowed for COMMON PILOT compatibility.

EXAMPLES:

```
PR:
P: UG
```

See also: ACCEPT, GOTO AND ESCAPE

POKE, OUT - set memory byte or do port output

PX:offset,value
PJ:address,value

The value must be a number or expression from 0 to 255. PX: stores the byte value in memory at the offset given. The offset is from a segment whose base is specified by the system variable %A. Great care must be taken in using this statement since a mistake could cause system failure. PJ: performs a machine level OUT instruction to the device address given of the byte value.

EXAMPLES

If you have both types of video adapters (monochrome and color) you can switch from the monochrome to the color for PILOT output via ...

```
C: %A=0
PX: 1040,pek(1040)-16
(then set desired mode via TS: statement)
```

and you can change back to the monochrome via...

```
C: %A=0
PX: 1040,pek(1040)+16
(then set desired mode via TS: statement)
```

REMARK - program comment

R: remarks

REMARK is ignored during program execution. It can be used to document a PC/PILOT program.

SOUND - generate tones

S: list

SOUND is used to generate simple tones, noises or beeps for audio feedback. The list contains one or more pitch or pitch/duration values. The entries in the list are separated by the ";" character. The two possible forms of a list entry are:

```
pitch
pitch, duration
```

where pitch is a number or variable from 0 to 2047 and duration is a number or variable from 1 to 63. A duration of 34 gives about a one second tone. The lower the pitch value the higher the tone. The higher the duration value, the longer the tone is played. If the duration is omitted after a pitch (first form above) then the duration of the previous pitch is used.

It is possible to repeat a series of notes by the notation:

```
*n(list)
```

where n is a number or a variable. A pitch value of 0 give silence. Normally, doubling a pitch lowers it by one octave. Halving a pitch raises it by one octave.

EXAMPLES:

```
S: 200,25;300,8;100,18

S: *4(208,5;224)

R: next two lines play a two-octave scale
S: 500,10;446;400;374;333;301;266;250
S: 250,10;223;200;187;166;150;133;125
```

√TYPE – display text on screen

P

T: text
TH: text
TX: text
: text

TYPE is used to display text and/or variables or expression values on the screen. The X (clear) modifier clears the text viewport to the current background color and homes the cursor prior to the text display. The text is displayed only within the current viewport with end-of-line wrap and scrolling as necessary. If the H (hang) modifier is omitted, the cursor is spaced down as specified by the current line spacing value after the text is displayed.

The text may contain a numeric or string variable preceded by a # or a $. The value of the variable is substituted in the text output. If the variable is a string or array it may be subscripted. If a space occurs after the variable name, the space is not displayed.

The text may contain an expression preceded by a # or $ and enclosed in parentheses. The value of the expression is substituted in the text output.

The text output may contain any printable character from CHR(32) to CHR(255). The following control characters may also be displayed by placing them in a string variable or embedded expression.

CHR(8)=BACKSPACE	CHR(9)=FORESPACE
CHR(10)=DOWN	CHR(11)=UP
CHR(12)=CLEAR	CHR(13)=NEWLINE
CHR(14)=HOME	

Note that DOWN and NEWLINE cause the viewport to scroll up if the cursor is on the bottom line.

UP causes the viewport to scroll down if the cursor is on the top line.

Any TYPE statement may be continued for as many lines as necessary by placing a colon in the first position of the next line.

EXAMPLES:

```
T: Good job $NAME.
: Your average is #(TOTAL/COUNT).

TX: This is on a clear screen.

TH: This is all
T: on one line.
```

See also: TYPE SCREEN

TYPE SCREEN - set screen options, animate

TS: list

TYPE SCREEN is used to control screen modes and colors, define viewports on the screen, set line spacing and make animated graphics. The list consists of one or more of the following commands. Each command is separated from the next by a ";" character. In the command descriptions below lower case letters denote a numeric variable or an integer constant.

Mn	-	set screen mode and erase screen. Possible modes are 0 through 6.
Ln	-	set spacing between text lines, n<=1 sets single spacing, n=2 sets double spacing, etc..
Fn	-	set foreground text color, possible values shown below.
Bn	-	set background color, possible values shown below.
En	-	set background color and clear the current viewport to the background color; cursor set to home position in viewport.
Xn	-	in modes 0-3 sets color of the border or outer frame of the screen, in mode 4-5 has same effect as Bn, in mode 6 sets the text color.
Vl,r,t,b	-	set screen viewport to be from column l on the left through column r on the right and from row t on the top through row b on the bottom. See below for more information on viewports.
V;	-	restore the previous viewport and cursor position.
Atext	-	animate; the text is displayed from the cursor position. If the text contains a "/" character, the "/" is not displayed but the next character is displayed on the next line down and directly under the original cursor location. Text is displayed up to a ";" or the end of the statement. After the text is displayed the cursor is left where it was when the animate began. This command is useful to draw figures made up of a block of several redefined characters.
Pn	-	display text page n. In modes 0 and 1 n may be 0-7, in modes 2 and 3 n may be 0-3 in modes 4-6 n may be 0 only. In the text modes (0-3) there are multiple display pages one of which is the currently visible page on the screen. The normal default is P0.
Dn	-	delay for n 60ths of a second. This is useful when using animate within a loop to move a figure on the screen.
Wrlud	-	walk right, left, up or down. The W can be followed by any number of r, l, u, or d characters. Each one moves the cursor one space in the indicated direction. Moving down on the bottom line of a viewport causes the viewport to scroll up; similarly moving up on the top line causes the viewport to scroll down.
Gx,y	-	position the cursor in column x and row y of the current viewport. G0,0 puts the cursor in the home position of the viewport.
*n(...)	-	repeats the commands in parentheses n times. Any of the above commands may be in the parentheses. This is useful to move a figure by repeating a sequence of animate, delay and walk in a loop.

VIEWPORTS

A viewport is a section of the screen which is used to display all text output by the program. When PC/PILOT starts the current viewport is defined to be the entire screen (V0,39,0,24). The Vl,r,t,b command defines a new section of the screen to be the current viewport. When this is done the cursor is placed in the home position of the viewport. All text displayed by the program displayed only in the current viewport, any data displayed outside the viewport is unaffected as the viewport changes, clears, scrolls, etc.. A V command without four numbers puts the viewport and cursor back to the position prior to the last V command. This simplifies alternating between two viewports on the screen for such uses as student responses and program feedback. Viewports are also useful to retain instructions, questions or graphs outside a viewport while conversation takes place within the viewport.

EXAMPLE:

TS:V5,30,18,25

SCREEN MODES

MODE	COLUMNS	ROWS	X-PIXELS	Y-PIXELS
0	0-39	0-24	-	-
1	0-39	0-24	-	-
2	0-79	0-24	-	-
3	0-79	0-24	-	-
4	0-39	0-24	0-319	0-199
5	0-39	0-24	0-319	0-199
6	0-79	0-24	-	-

COLORS IN MODES 0-3

0 - BLACK	1 - BLUE	2 - GREEN
3 - CYAN	4 - RED	5 - MAGENTA
6 - BROWN	7 - LIGHT GREY	

FOREGROUND: can be set to any of the above, on some monitors adding 8 to above number gives lighter version of same color.

BACKGROUND: can be set to any of the above, adding 8 above numbers also causes any characters written to blink in modes 0-3. In modes 4 and 5 adding 8 gives lighter version of the same color.

COLORS IN MODE 4

0 - current background color
1 - GREEN 2 - RED 3 - YELLOW

COLORS IN MODE 5

0 - current background color
1 - CYAN 2 - MAGENTA 3 - WHITE

COLORS IN MODE 6

Mode 6 is monochrome, white on black.

ANIMATION

This example shows how to move a figure across the screen left to right then right to left. The figure is a block of four letters:

WX
YZ

but it could be any user defined characters as well. Notice that the animation string is carefully laid out with spaces to insure that each redraw of the figure completely overtypes

the previous one so a trail is not left behind. Also note that the number after the D controls the speed of motion.

```
TS:M1;B0;F1;*20(A WX/ YZ;WR;D5)
TS:*20(WL;AWX /YZ ;D5)
```

Note: The various modes and colors apply only to the color adapter card. On monochrome adapter the screen always works in 25 by 80 mode.

See also: TYPE, GRAPHICS

SR

✓ USE - call a subroutine

U: destination

USE jumps to the destination and saves the location of the statement after the USE on the use-stack. A subsequent END statement in the subroutine causes a return to the statement after the USE. The destination can be any of those described for JUMP although the only one which normally makes sense is a statement label.

USE is similar to gosub in basic.

EXAMPLE:

```
U:GLOSS
```

See also; END, JUMP

VIDEO - call external driver program

V: variable$

VIDEO is intended to allow a PC/PILOT program to call a machine language subroutine to control a video disk or video tape. However, it could be used for many other purposes since it is actually a general purpose interface to any machine language program.

The only argument of a VIDEO statement is the name of a string variable. The string variable must have been dimensioned long enough to contain the machine language subroutine, and the subroutine must have been placed into the string. The DX: op code must be used to dimension the string variable. This insures that the string begins on a 16-byte boundary.

The VIDEO statement causes a machine language FAR CALL to be executed to the first byte of the string. Upon entry to the subroutine the code segment and data segment registers are equal and point to a segment which contains the subroutine. Segment registers must not be modified upon return from the subroutine. Other registers may be used without saving the contents. The subroutine must return via a FAR RETURN op-code. Data can be passed to/from the subroutine by the PC/PILOT program in mutually decided portions of the string variable itself. The PILOT program can use subscripting to deposit or fetch data in the locations within the string designated by the subroutine.

A good way to get the subroutine into the string variable is to read it from a disk file as shown in the first example below. The machine language code must be in memory image format and must be position independent since it may be loaded anywhere. Naturally, care should be taken in using this statement since improper use

can cause system failure. The DOS utility called EXE2BIN may be of use in preparing the subroutine file.

EXAMPLE:

```
DX: SUB$(1000)
FX: VIDSUB.COD
FI: 0,SUB$
V: SUB$
```

See also: sample program SAMPLE4.PIL

T

√ WAIT - program delay

W: expression

WAIT causes a program delay for expression tenths of a second. The delay is prematurely ended if the student pushes a key. Thus the example below waits for ten seconds or until a key is pushed, whichever occurs first.

EXAMPLE:

```
W:100
```


J

EXECUTE INDIRECT - execute a string

XI: expression

EXECUTE INDIRECT evaluates the expression to its string value. Then the string value is executed as if it were a PC/PILOT statement. The string may contain any valid statement up to length 80. If the string is too long only the first 80 characters are used. The "I" of "XI" may be omitted; it is allowed for COMMON PILOT compatibility.

EXAMPLES:

```
R: random selection of a problem
X: "J:PROB"!!RND(10)

R: store a graphic in a string to REUSE it
C: BOX$="G:*4(F50;R90;F100;R90)"
XI: BOX$
```

EXPRESSIONS

PC/PILOT accepts expressions in many contexts. An expression is a combination of variables, numeric constants, string constants, functions and operators. Expressions are formed in the normal manner allowed by other programming languages. Operator precedence is similar to that of basic. Parentheses may be used to group subexpressions. Each element of an expression is a number or a string value. Numbers are stored in floating point form. Strings are stored in variable length form with a terminating null character. Each operator or function expects to act on a particular type of argument. And in turn produces a particular type of result. For example, + expects two numeric arguments and it produces a numeric result. A unique and convenient feature of PC/PILOT is auto-type conversion. This means that any time an argument is of the wrong type (string or number) it is automatically converted to the correct type for the context. A number is converted to a string in a manner like that of the STR function, which puts the value in printable format with decimal places only if necessary. A string is converted to a number in a manner like that of the FLO function, which scans the string for the first number value or returns zero if no number is found.

EXAMPLES:

```
C: X = 4*(A-B / (E))

C: WORDS$ = CAP(LIST$(I,J))
```


VARIABLES

A variable name can be up to length six. The first character must be a letter. Other characters may be letters or digits. The last character may be a $ to signify a string variable. Upper and lower case letters are considered to be equal when naming a variable (i.e. - XYZ and xyz name the same variable). A string variable must be dimensioned in a D: statement before it can be used to store a string value. There are two system variables %A and %B which can be used like other variables. %A is a number equal to the number of times the last ACCEPT has been executed without any intervening other ACCEPTS (i.e. - how many trys the student has made on this question). %B is a string variable which is set to the response given by the student on each ACCEPT.

EXAMPLES:

```
I3
name$
COUNT
```

See also: DIMENSION, SUBSCRIPTING

SUBSCRIPTING

array (position)
string$ (position)
string$ (position, length)

SUBSCRIPTS are used to select an individual value out of a numeric array, or a substring out of a string variable. Before subscripts can be used the variable must have been dimensioned. Numeric arrays start at subscript zero and end at the dimensioned size. String subscripts start at one. When subscripting a string, if the length is omitted, a length of one is assumed. An attempt to subscript beyond the current length of a string gives an error. In general, a subscripted variable may be used anywhere a simple variable may be used. Position and length values may be expressions. A reference to an array variable without subscripts references element zero. The system variable %B is a string and may be subscripted as such.

EXAMPLES:

```
A(I) = A(I+1) * 3

J(B$(I) = "A"): LABEL

T:$HELP$(4,10)
```

See also: DIMENSION, VARIABLES

NUMERIC CONSTANTS

[-]digits
[-]digits.digits

A numeric constant consists of an optional leading minus sign, a string of decimal digits optionally including a decimal point.

EXAMPLES:

```
5
63.789
-42.1
```

STRING CONSTANTS

"text"

A string constant, or literal consists of zero or more characters enclosed in quotes.

EXAMPLE:

```
A$="BANANA"
T(X$ < "A"): NO SIR.
```


ARITHMETIC

```
x + y     x - y     x * y     x / y
x % y  (remainder of x/y)        -x
```

To compute A to the B power use EXP(B*LNE(A))

RELATIONAL

```
x < y     x > y     x = y     x <> y
x <= y    x >= y
```

Comparison is numeric if x is a number, or string if x is a string. For string comparisons the shorter string is padded on the right with spaces for comparison purposes. Relational operators give 1 for true or a zero for false.

LOGICAL

```
x & y     (true if both x and x are non-zero)
x ! y     (true if either x or y is non-zero)
^x        (true if x is zero)
```

Logical operators always produce a 1 for true or a zero for false.

STRING

x$!! y$ (concatenation)

Concatenation produces a string consisting of the string value of y$ appended to the right end of the string value of x$.

FUNCTIONS

ABS(X)	the positive value of X
ASC(X$)	the numeric value 0-255 of char X$
ATN(X)	arctangent in degrees of X
AUX(0)	reads the com1: (rs232) port, returns

	CHR(0) if no byte ready to read, or a character from CHR(1) to CHR(255).The coml: port must be set up via a MODE command before entering PILOT.
CAP(X$)	the upper case value of string X$
COS(X)	cosine of X degress
CHR(X)	returns the Xth ascii character
EXP(X)	e to the X power
FLO(X$)	the numeric value of the first number found in string X$
INS(X$)	zero if character X$ is not in %B, otherwise the first position in %B where X$ is found
INP(X)	machine level IN instruction, returns a byte from io port X
INT(X)	value of X truncated to an integer
KEY(X)	zero if no key is pressed, otherwise the ascii code for the depressed key
LEN(X$)	current length of string X$
LNE(X)	natural log (base e) of X
LOG(X)	log base 10 of X
PEK(X)	returns a value from 0 to 255 equal to the byte in memory at offset X in the segment whose base is in the system variable %A
RND(X)	if X is zero returns a random fraction from 0 to 1. If X>0 returns a random integer from 0 to X-1.
RSP(X$)	the string value of X$ with all spaces removed
SIN(X)	sine of X degrees
SQR(X)	square root of X
STR(X)	the string value of number X
SWP("xy")	returns a string equal to the value of %B except that each character x is changed to character y

EXAMPLE:
```
R:read N characters from rs-232 port
D:X$(80)
C:X$ = ""
*LOOP  C:X$= X$ !! AUX(0)
J(LEN(X$)<N):LOOP
```


GOTO AND ESCAPE - run time commands

GOTO destination

@ any text

Each of these commands can be enabled or disabled by the PROBLEM statement. If enabled then every time an ACCEPT statement is executed PC/PILOT automatically checks the student response for the presence of the command. A GOTO command is signified by the word goto followed by a destination. It causes an immediate jump to that destination. It is useful during program testing to jump around in a program and check out various sections. It can also be used to allow the student to move around in the program. The ESCAPE command is signified by a response which begins with the "@" character. It causes an effect equivalent to the statement U:SYSX . That is, a subroutine named SYSX is called. It is up to the program author to insure that label SYSX exists. The SYSX routine could be used to record comments from the student as shown in the first example below. It could also be used to allow the display of a menu or glossary on demand. The SYSX routine should end with an END statement. If no argument is given on the END then a return is made to the statement after the ACCEPT. The first example below returns to re-execute the ACCEPT. Note also that since the END can specify a label, it is not necessary to return at all, as shown in the second example.

EXAMPLES:

```
*SYSX  K:%B
 E:@A
```

```
*SYSX  E:SYSX2
*SYSX2  T: GOING TO MAIN MENU.
J:MENU
*SYSX  R: DISPLAY GLOSSARY SCREEN
TS:P3
R: WAIT 30 SECS OR UNTIL KEY HIT
W:300
R: PUT BACK ORIGINAL SCREEN
TS: P0
R: RETURN AND REACCEPT
E: @A
```

See also: PROBLEM, ACCEPT, USE, END

ERROR MESSAGES

PC/PILOT handles programming errors in forgiving way. If an error is encountered th statement in error is displayed on the screer Before it is a message which denotes what th problem is. Execution then pauses. The use can push CTRL-C to stop the program or can pus any other key to ignore the error and continu execution with the next statement. If it is a all possible to go on, and it usually is PC/PILOT will continue with the program. Th following error codes are used by PC/PILOT.

disk	- i/o error in program file
exp	- invalid or missing expression
file	- no file open or disk i/o error
label	- missing destination
link	- program file not found (program stops)
lspace	- too many labels in one program module
mode	- graphics done when not in mode 4 or 5
open	- can not open disk file
opcode	- invalid op code or modifiers
paren	- missing parenthesis
quote	- missing closing quote
redim	- attempt to dimension a string or array which is already dimensioned
sspace	- out of string scratch pad space
subscript	- isto big for the array or current length of the string
syntax	- statement has bad command or operator
uspace	- use call level is too deep (over 5)
var	- missing variable name
vspace	- too many variables in use
xspace	- no more memory for arrays or stings

EZ EDITOR

EZ is a simple screen oriented editor. To use it in 80 column mode enter

```
x:EZ  y:name.PIL
```

or to use it in 40 column mode

```
x:EZ40  y:name.PIL
```

where x: and y: are the normal optional path or drive designations. If the named file does not exist it is created. If it does exist, it is read in and displayed on the screen. If any line is too long to fit across the screen (i.e. - over 79 or over 39), it is split into two lines by inserting a new-line sequence. Upon exit a backup copy of the file is retained as name.BAK. Once in EZ the screen is a window on the text file. The various cursor and function keys are used to move around in the text and to add, change or delete text. If the entire text file does not fit in memory then you can tell EZ to put some lines out to the result file and load in more lines to work on. When adding or replacing text you just type on the keyboard what you see on the screen is what you get in the text file. The following summary shows the various keyboard functions you can use.

UP, DOWN, RIGHT and LEFT arrows	move the cursor one space in any specified direction.
TAB	move the cursor to the next column which is a multiple of 10.
HOME	move to the first text line currently in memory.
END	move to the last text line currently in memory.
PG UP	move up (backwards) one screenful in the text.